Rapport de stage

Développement d'une plateforme de tests automatisés : GreenT

Antoine de Roquemaurel

M1 Informatique – Développement Logiciel
2014 – 2015

Maître de stage :
Alain Fernandez

Tuteur universitaire :
Bernard Cherbonneau

Du 04 Mai au 28 Août 2015
Version du 16 août 2015

Je tiens à remercier toutes les personnes m'ayant permis de réaliser ce stage.

En premier lieu, un grand merci à Corinne Tarin pour m'avoir accepté au sein de son équipe.
Je remercie particulièrement Alain Fernandez pour m'avoir suivi et conseillé tout au long de ce stage tout en partageant son expérience.
Une pensée à Joelle Decol pour sa bonne humeur quotidienne, ainsi qu'à toute l'équipe du troisième étage, grâce à qui j'ai passé d'excellents moments au sein de l'entreprise.
Merci à mon tuteur universitaire Bernard Cherbonneau pour son suivi et sa visite en entreprise.
Enfin, je remercie toutes les personnes m'ayant entouré durant ce stage et aidé à la rédaction ce rapport, à savoir Diane, Ophélie, Clément et Mathieu.

Introduction

Dans le cadre de ma formation en première année de Master spécialité Développement Logiciel à l'université Toulouse III – Paul Sabatier, j'ai eu la possibilité d'effectuer un stage.

Attiré par le monde de l'entreprise et désireux de gagner en expérience, j'ai eu l'opportunité de continuer un projet commencé l'année précédente lors de mon stage de fin de Licence dans l'entreprise Continental Automotive : le développement d'une plateforme de tests de logiciels embarqués.

Ce projet a pour but d'aider une équipe de Continental, celle-ci est en charge de l'intégration d'un plugin – fourni sous forme binaire – dans le logiciel d'un calculateur de contrôle moteur. Pour cela, une plateforme permettant d'effectuer des tests automatisés est en développement.

Ayant connu les prémices de ce projet, et afin d'avoir un aperçu de celui-ci sur la durée, allant de sa conception jusqu'à sa mise en production, le sujet du stage était particulièrement intéressant. En outre, celui-ci est en parfaite adéquation avec mon projet professionnel, et ma poursuite en M2 Développement Logiciel.

C'est au sein de l'équipe *Test & Automation Service* que j'ai effectué mon stage d'une durée de quatre mois, je vais ainsi vous présenter en quoi le développement de cet outil est nécessaire à l'équipe en charge des tests de ce plugin. Dans une première partie nous présenterons l'entreprise Continental et plus particulièrement l'équipe Tests & Automation Service(chapitre 1), puis nous verrons de quelle manière nous nous sommes organisés pour le développement (chapitre 2). Nous aborderons ensuite le problème que posent actuellement les tests de ce plugin(chapitre 3), avant de présenter la solution qui est en cours de développement(chapitre 4) et comment j'ai contribué à ce projet(chapitre 5).

Table des matières

Remerciements **3**

Introduction **5**

1 Continental **9**

1.1 Organisation de l'entreprise 9

1.2 Le contexte de l'équipe TAS 12

2 Organisation du travail **13**

2.1 L'équipe de développement 13

2.2 La documentation : les *minutes étude* 13

2.3 Outils de développement 14

3 Le problème **17**

3.1 Les tests actuels 17

3.2 La solution : *GreenT* 19

4 *GreenT* : fonctionnement général **21**

4.1 Le fichier Walkthrough 21

4.2 Fonctionnement Global 22

4.3 Les différentes fonctionnalités du Client 24

4.4 Fonctionnement des serveurs 27

5 Ma collaboration au projet **29**

5.1 Les calibrations 29

5.2 Le « patch calib » 29
5.3 Les « tableaux calibrables » 29
5.4 La maintenance 29

6 Bilans **33**
6.1 Bilan pour Continental 33
6.2 Bilan personnel 33

A Acronymes et Glossaire **37**

B Les différents équipements **39**
B.1 Le HIL DSpace 39
B.2 Le Debugger 40

C Références **41**
C.1 Documentations en ligne 41
C.2 Livres 41
C.3 Cours magistraux 41
C.4 Sites Web et forums 42

D Exemple de rapport généré par GreenT **43**

E Exemples de fichiers générés **45**
E.1 Exemple de StimScenario 45
E.2 Exemple de `GreenTTest` 47

F Liste des codes sources **49**

G Table des figures **51**

1 Continental

J'effectue mon stage au sein de l'entreprise Continental, une entreprise ayant pris une très grande importance dans le monde de l'automobile.

Sommaire

1.1 Organisation de l'entreprise 9
1.2 Le contexte de l'équipe TAS 12

Afin de vous présenter le contexte dans lequel j'ai travaillé, nous allons voir comment opère cette société, et plus particulièrement l'équipe dans laquelle j'ai travaillé : l'équipe *Tests & Automation Service*.

1.1 Organisation de l'entreprise

1.1.1 Continental AG

Continental AG est une entreprise allemande fondée en 1871 dont le siège se situe à Hanovre. Il s'agit d'une Société Anonyme (SA) dont le président du comité de direction est le Dr. Elmar Degenhart depuis le 12 août 2009. Elle est structurée autour de deux grands groupes : le groupe Rubber et le groupe Automotive.

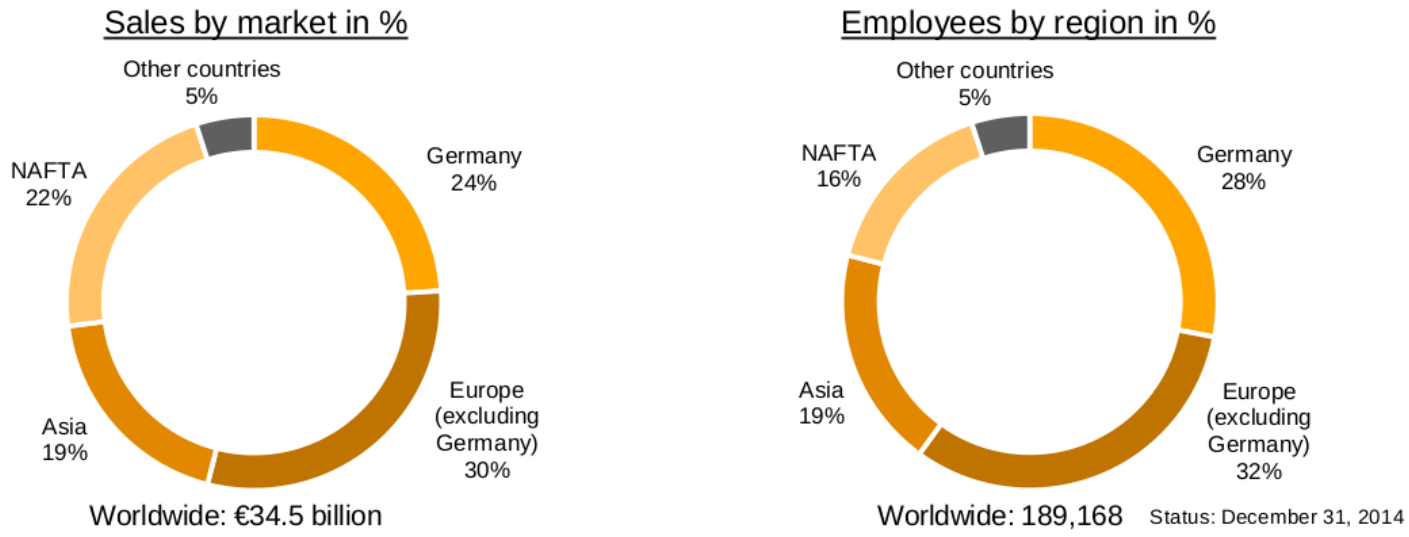

Figure 1.1 – Chiffre d'affaire et nombre d'employés (Année 2014)

En 2014, l'entreprise comptait plus de 189 000 employés dans le monde[1] répartis dans 317 sites et 50 pays différents[2]. Avec un chiffre d'affaire de 34.5 milliards d'euros au total, Continental est le numéro un du marché de production de pneus en Allemagne et est également un important équipementier automobile.

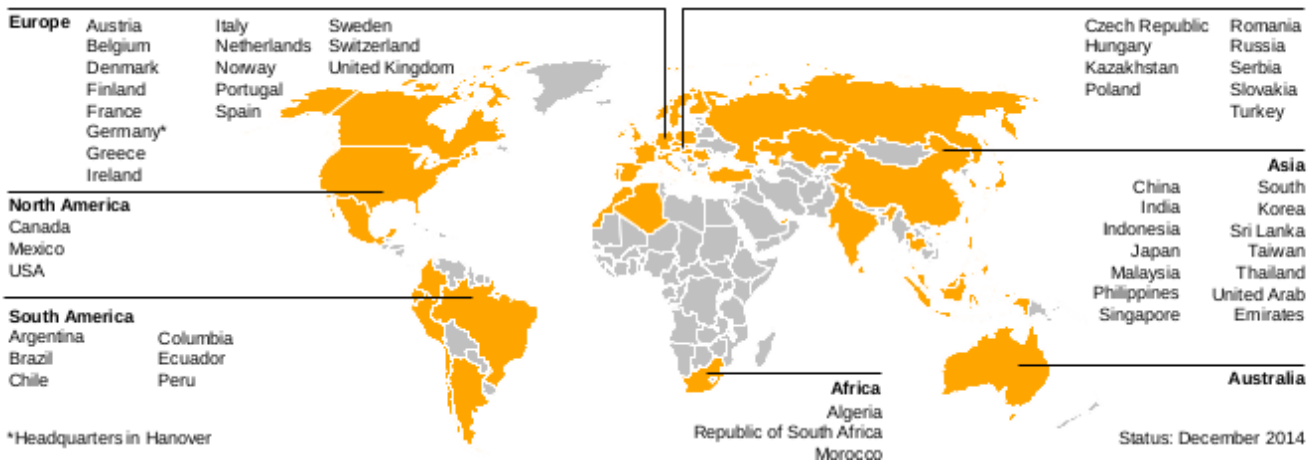

Figure 1.2 – Répartition du groupe Continental dans le monde

1.1.2 Histoire de l'entreprise

Continental est fondée en 1871 comme société anonyme sous le nom de « *Continental-Caoutchouc- und Gutta-Percha Compagnie* » par neuf banquiers et industriels de Hanovre (Allemagne).

Continental dépose l'emblème du cheval représenté sur la figure 1.3, comme marque de fabrique à l'Office impérial des brevets de Hanovre en octobre 1882. Ce logo est aujourd'hui encore protégé en tant que marque distinctive.

Figure 1.3 – Logo de Continental

Le fabricant de pneus allemand débute son expansion à l'international en tant que sous-traitant automobile international en 1979, expansion qu'il n'a cessé de poursuivre depuis.

Entre 1979 et 1985, Continental procède à plusieurs rachats qui permettent son essor en Europe, celui des activités pneumatiques européennes de l'américain *Uniroyal Inc.* et celui de l'autrichien *Semperit*.

1. Cf figure 1.1
2. Cf figure 1.2

En 1995 est créée la division « *Automotive Systems* » pour intensifier les activités « systèmes » de son industrie automobile.

La fin des années 1990 marque l'implantation de Continental en Amérique latine et en Europe de l'Est.

En 2001, pour renforcer sa position sur les marchés américain et asiatique, l'entreprise fait l'acquisition du spécialiste international de l'électronique *Temic*, qui dispose de sites de production en Amérique et en Asie. La même année, la compagnie reprend la majorité des parts de deux entreprises japonaises productrices de composants d'actionnement des freins et de freins à disques.

En 2004, le plus grand spécialiste mondial de la technologie du caoutchouc et des plastiques naît de la fusion entre *Phoenix AG* et *Conti'Tech*.

Enfin en juillet 2007, Continental réalise sa plus grosse opération en rachetant le fournisseur automobile *Siemens VDO Automotive*. Ce rachat a permis à l'entreprise de multiplier son chiffre d'affaire par deux, passant ainsi de 13 milliards d'euros à plus de 30.5 milliards d'euros (chiffre de 2014).

1.1.3 Activités des différentes branches

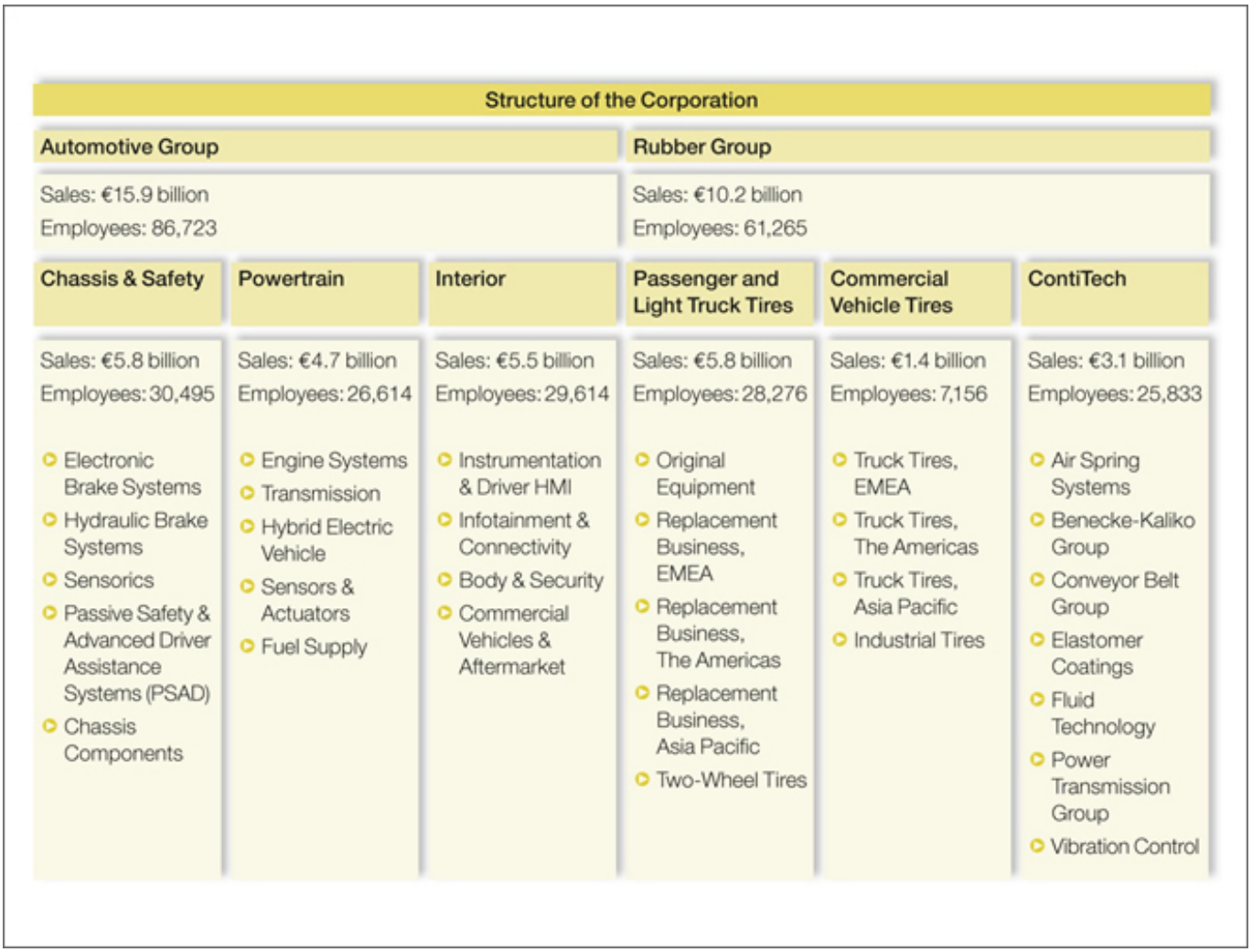

FIGURE 1.4 – Structure de Continental

Comme on peut le voir sur la figure 1.4, Continental est composée de deux groupes et de six divisions. Ces dernières se chargent de développer et produire des équipements répondant aux besoins des clients. Pour cela elles sont composées de *Business Units* qui ont chacune une activité bien particulière dans leur domaine de compétence.

Durant mon stage, je travaillais au sein de *P-ES* : division *Powertrain* et *Business Unit Engine Systems*. Elle s'occupe essentiellement du contrôle moteur, au niveau logiciel et matériel avec l'ECU [3] et de la mise au point des systèmes diesel et essence. La BU *Engine Systems* est quant à elle chargée de produire les équipements nécessaires au contrôle moteur tels que des calculateurs ou des injecteurs.

1.2 Le contexte de l'équipe TAS

1.2.1 L'équipe Tests & Automation Service

J'ai travaillé dans l'équipe en charge des tests au niveau système ou logiciel dirigée par Corinne Tarin.

Pour ma part, j'opérais dans la partie logicielle des tests, « sous-équipe » ayant en charge le développement, la configuration et l'exécution de scripts de tests de non-régression automatique [4] sur bancs HIL [5] avant la livraison des projets.

1.2.2 Le besoin

Le calculateur du contrôle moteur d'une voiture est un dispositif très important et à haut risque, en effet, une défaillance peut provoquer la mort de plusieurs personnes [6]. Ainsi, le test est indispensable dans ce domaine, et doit être robuste.

L'automatisation des tests est rendue nécessaire pour deux raisons. Tout d'abord, un logiciel ne peut comporter aucun bug, cependant les erreurs et bugs critiques liés à l'inattention peuvent être grandement réduits grâce à ce processus. De plus, au vue de la complexité d'un contrôle moteur, et de son nombre d'instructions, une opération manuelle serait impensable.

C'est dans ce contexte que l'équipe *Tests & Automation Service* intervient, elle doit fournir des outils aux développeurs afin de vérifier facilement et correctement leur travail, particulièrement pour des tests de non-régression, bien que l'outil sur lequel je travaille soit à destination de tests d'intégration.

3. Electronic Control Unit
4. Aussi appelés FaST : **F**unctions **a**nd **S**oftware **T**esting
5. **H**ardware **I**n the **L**oop, vous trouverez plus d'explications sur ce dispositif section 3.1.1
6. Le programme d'une voiture comporte ainsi des fonctions dites « *safety* » tel que le régulateur, l'accélération, le freinage, ...

2 Organisation du travail

Étant donné la complexité du projet et son importance, une organisation réfléchie est indispensable. Autant d'un point de vue humain, avec une gestion de projet et une gestion de l'équipe, que technique en utilisant certaines technologies nous aidant dans la tache. Nous allons voir l'organisation qui a été mise en place afin d'être le plus efficace possible.

Sommaire

2.1 L'équipe de développement 13
2.2 La documentation : les *minutes étude* 13
2.3 Outils de développement 14

2.1 L'équipe de développement

Au cours de mon stage, trois développeurs travaillaient sur le projet *GreenT* : Alain Fernandez, chef d'équipe et membre de l'équipe *Tests & Automation Service*, Benjamin Guerin, apprenti, et moi-même, stagiaire au sein de la même équipe.

En tant que chef d'équipe, Alain Fernandez organisait les réunions et supervisait notre travail tout en corrigeant des bogues, et validait manuellement les rapports de tests [1]. Benjamin Guerin se chargeait d'effectuer une étude de faisabilité sur l'amélioration visuelle des rapports de tests. Quant à moi je m'occupais de développer deux nouvelles fonctionnalités, à savoir les tableaux calibrables et le patch de calibration, tout en corrigeant différents bogues. [2].

À mon arrivée, la plateforme était quasiment opérationnelle, grâce à notre travail lors de mon précédent stage, et de l'avancement qui avait été fait durant cette année de césure. Je me suis donc d'abord renseigné sur les différentes améliorations et avancées du développement afin d'être rapidement opérationnel.

Ensemble, nous avons convenu de documenter au maximum notre travail, afin de conserver une plateforme toujours à jour au niveau de sa documentation. Ainsi pour chaque modification, chacun de nous devait remplir un document de *minutes d'étude*.

2.2 La documentation : les minutes étude

Je suis arrivé en Mai sur un projet ayant été commencé 18 mois auparavant, ainsi beaucoup de choses existaient déjà : il était nécessaire de garder l'existant. Afin de documenter notre travail,

1. La validation manuelle des rapports est indispensable afin de vérifier que notre plateforme fournit des rapports fiables
2. Cf chapitre 5

il a été décidé que pour chaque développement, que ça soit du bogue, de la fonctionnalité ou de la réorganisation du code, il était nécessaire de remplir un document Word de *Minutes étude*. Ce document comportait 4 grandes parties :

Analyse du besoin Pourquoi ce développement est nécessaire, les cas d'utilisation pris en charge, ou non, les éventuelles discussions avec l'équipe cliente.

Analyse de l'existant Retro-engineering permettant de comprendre le fonctionnement actuel du module que nous allons modifier, cette conception doit être statique, et dynamique, appuyé sur des schémas UML 2.

La solution Conception de notre solution, ses limites, ses avantages. De la même manière que la partie précédente, la conception est statique et dynamique, avec des schémas UML 2.

Les tests De quelle manière nous allons tester notre solution, aussi bien en tests unitaire qu'en tests d'intégration.

Ces différents documents pourront nous resservir plus tard pendant la maintenance, si un modèle doit être amélioré, ou comporte des problèmes, la relecture de la minute étude correspondante nous fera gagner beaucoup de temps.

2.3 Outils de développement

Afin de travailler de façon efficace, nous avons utilisés des outils aidant au développement.

2.3.1 Java

À mon arrivée, la partie client de notre plateforme était développée en Java dans sa version 6.0, Java nous permettant d'avoir un langage fortement typé, très puissant au niveau du paradigme Objet, connu de l'équipe, assez simple de déploiement et multiplateforme.

Une de mes collaboration a été le passage à Java 8 nous permettant d'utiliser toute la puissance de Java, et d'avoir une plateforme qui soit à jour au niveau technologique.

2.3.2 Git

Nous avons utilisé *Git* afin de faciliter le travail collaboratif d'une part, et de versionner le code du logiciel d'autres part. Git permet de fusionner les modifications de plusieurs développeurs, tant que nous ne modifions pas le même fichier en même temps. Ainsi, la fusion de nos modifications était faite automatiquement.

De plus, à chaque nouvelle modification, un « commit », permet de créer un point de restauration : il est alors possible de récupérer n'importe quelle version du logiciel depuis son commencement. Nous y insérons un message clair expliquant ce qui a été fait, cela permet aux autres développeurs de l'équipe de se tenir au courant de l'avancement.

2.3.3 Eclipse

Nous développions tous sous le même environnement de développement Eclipse, avec le plugin *Git* et le plugin *PyDev*. Le plugin Git permet d'avoir des outils aidant à la résolution d'éventuels conflits et le plugin PyDev permet de développer avec l'interpréteur et la coloration syntaxique Python.

2.3.4 Thrift et client-serveur

Notre plateforme fonctionne avec une architecture client-serveur, un client et deux serveurs. Le client écrit en Java, un serveur utilise Python et le second est lui aussi en Java. Afin de faire communiquer les deux parties de notre application, nous avons utilisé *Apache Thrift*. Il s'agit d'une bibliothèque ayant pour but les communications réseau inter-langage, dans le même principe que le protocole RMI [3].

2.3.5 UML et Entreprise Architect

Nous avons travaillé avec la norme UML [4] 2 afin de concevoir la plateforme, en utilisant particulièrement des diagrammes de classes, mais aussi des diagrammes de cas d'utilisation ou d'activité.

Pour dessiner ces diagrammes, et les noter dans la documentation, nous les pensions d'abord sur tableau blanc, mais ensuite nous avions besoin d'un outil puissant afin de les dessiner sur informatique. Pour cela nous avons utilisé *Enterprise Architect*, un logiciel propriétaire permettant de créer tous les diagrammes de la norme UML 2.

2.3.6 LaTeX

Afin de rédiger ce rapport, et le diaporama de soutenance, j'ai utilisé LaTeX, un langage et un système de composition de documents fonctionnant à l'aide de macro-commandes. Son principal avantage est de privilégier le contenu à la mise en forme, celle-ci étant réalisée automatiquement par le système une fois un style défini.

LaTeX

3. Remote Method Invocation
4. Unified Modelling Language

3 Le problème

Depuis longtemps, l'entreprise avait un problème afin d'effectuer des tests d'intégrations, notamment pour les projets à destination de Ford. Les tests demandaient du temps et de l'argent à l'équipe en charge de ces tests. Ainsi, deux ans avant mon stage de M1, une solution à été trouvée : le développement d'une nouvelle plateforme, *GreenT*.

Sommaire

3.1 Les tests actuels 17
3.2 La solution : *GreenT* 19

3.1 Les tests actuels

Comme expliqué dans la section 1.2.2, le calculateur moteur est un système critique, il est donc indispensable de tester correctement celui-ci.

3.1.1 Les bancs de tests

Afin de tester au mieux les programmes du contrôle moteur développés, ceux-ci sont d'abord testé via des simulateurs d'environnement véhicule. Ce simulateur permettent d'une part de vérifier la bonne conformité entre le matériel et le logiciel, mais permettent également de vérifier le programme avant d'effectuer des tests sur véhicule.

Comme vous pouvez le voir figure 3.1, un banc de tests est composé d'un ordinateur, du calculateur appelé ECU pour ***E**lectronic **C**ontrol **U**nit* et d'au moins deux équipements aidant aux tests.

Les deux *équipements* étant les suivants :

Le HIL Le ***H**ardware **I**n the **L**oop* est un simulateur d'environnement véhicule. Ainsi l'ECU est branché sur le HIL et se comporte de la même manière que s'il était embarqué dans une voiture. Le HIL quant à lui est chargé d'envoyer les bons stimulus sur les pins de l'ECU, tel que l'injection, la vitesse de rotation du moteur, le starter ...

Debugger Cet *appareil* est branché sur l'ECU est va être capable d'effectuer un certain nombre d'actions directement sur l'ECU. Tel que flasher le logiciel à tester, mettre des points d'arrêts sur le code, lire des variables, les modifier, changer des calibrations, ...

> Un certain nombre d'équipes projet chez Continental utilise un troisième outil qui n'est pas représenté ici, parce que notre plateforme ne s'en sert pas. Cet outil, nommé INCA, va interagir sur l'ECU via un bus CAN [a]. Cet outil étant sujet à contreverses, il a été choisi de ne pas l'intégrer à notre plateforme.
>
> *a.* Controller Area Network

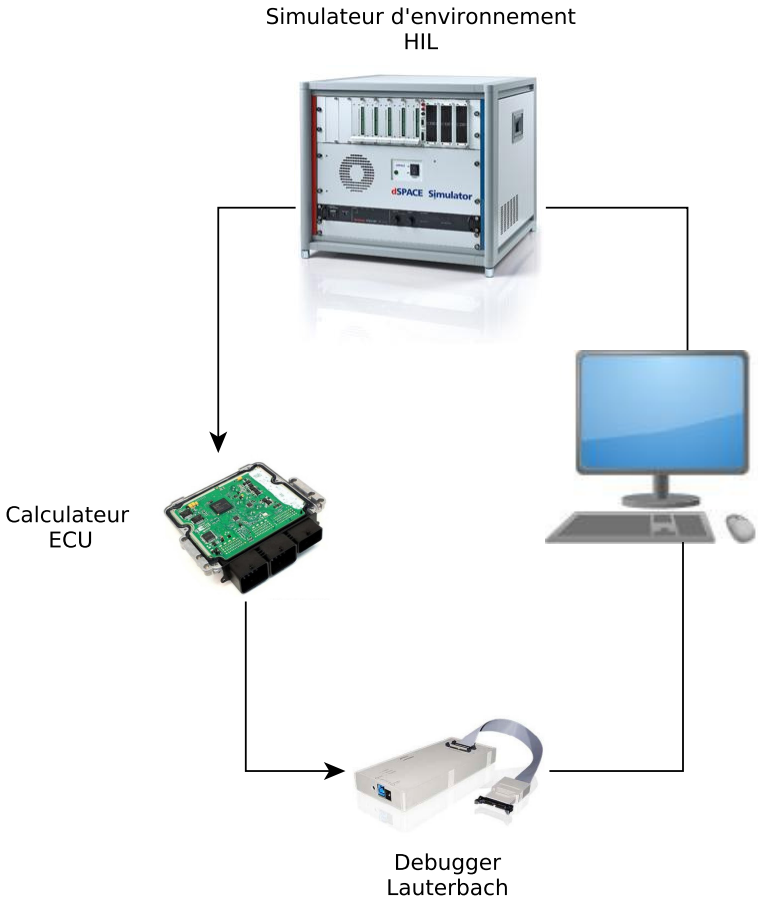

Figure 3.1 – Fonctionnement d'une table de tests : HIL DSpace, Debugger et ECU

3.1.2 Le plugin Ford

Dans le cadre de projets pour Ford, Continental ne développe pas l'intégralité du logiciel, en effet une partie est fournie par le client sous forme de « plugin ». Le *plugin* est supposé correct, et le tester n'est pas de notre ressort. Cependant, celui-ci va être interfacé avec les logiciels Continental : il est indispensable de vérifier que les deux parties fonctionnent ensemble lors de l'intégration.

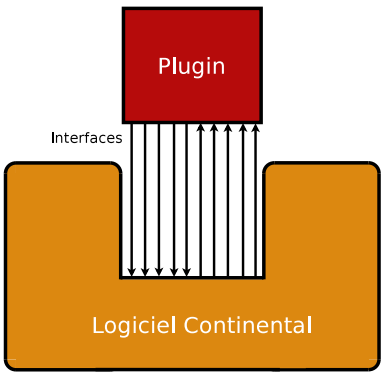

Figure 3.2 – Interfaces du plugin avec le logiciel de Continental

> Continental n'a pas accès au code source de ce plugin, l'équipe n'aura que le binaire du logiciel ainsi que sa spécification.

Pour cela, le client fourni un fichier appelé `Walkthrough`[1] contenant la liste des variables du plugin avec toutes leur spécifications, ce fichier est au format *Excel* : il contient environ 900 variables différentes. Il est impensable de tester le fonctionnement d'autant de paramètres manuellement. Ainsi l'équipe en charge de tester cette intégration effectue des tests de différence d'une version à l'autre : seules les variables ayant pu être impactées par une *release* seront testées, il est supposé que le fonctionnement des autres variables reste inchangé.

Trois problèmes se posent à cette méthode :

La fiabilité des tests manuels Le test des seules différences ne permet pas nécessairement de détecter tous les problèmes (notamment avec des effets de bords...). De plus, une tâche répétitive peut entrainer des erreurs humaines.

Le temps de tests Même en ne testant qu'une partie des variables, cela prend un temps considérable, il faut compter environ une semaine.
Or, les tests s'effectuent sur les bancs de tests comme expliqué section 3.1.1, ces équipements permettent de simuler un environnement voiture autour du contrôleur moteur comme l'utilisation de la clé de démarrage, la tension de la batterie, la vitesse de rotation du moteur, ... Ces bancs sont peu nombreux dans l'entreprise en raison de leur cout, leur disponibilité est compliquée. Il serait intéressant de pouvoir lancer des tests automatisés durant la nuit par exemple afin de les décharger de ce travail.

3.1.3 La plateforme TA3

Actuellement, les équipes de tests disposent d'une plateforme appelée TA3. Celle-ci est une bibliothèque de classes écrites en Python. Jusqu'à présent, pour chaque objectif de test, il fallait écrire un script python utilisant la TA3. Ces scripts pilotent le banc HIL et le debugger afin d'envoyer des stimulis à l'unité de contrôle moteur et de vérifier que les réactions de celui-ci sont conforme aux spécifications de test.

Cependant, cette plateforme pose un certain nombre de problèmes qui rend son utilisation difficile. D'une part, elle renvoie un trop grand pourcentage de faux-positifs, faisant perdre du temps au testeur. D'autres part, elle ne prend pas en compte certains besoins apparu récemment comme par exemple un système permettant de flasher automatiquement les ECU[2], ou la possibilité de vérifier la fréquence de mise-à-jour de la production de variables[3].

Afin d'améliorer cette situation, l'équipe *Tests & Automation Service* développe une nouvelle plateforme.

3.2 La solution : GreenT

Afin de résoudre les problèmes présentés dans la section 3.1, une solution a été pensée en étudiant les besoins de l'équipe en charge des tests du plugin : le développement d'une plateforme de tests, appelé *GreenT*.

1. Ce fichier est expliqué plus en détail section 4.1
2. Cela permettrai de scripter un test qu'on lancerai plus tard et de gagner du temps
3. Ceci afin de contrôler le temps réel du calculateur

> Le nom de *GreenT* provient de la contraction de *Green* et *Test*.
>
> En effet, un test est signalé correct si celui-ci est vert, or le but de notre plateforme est d'automatiser des tests et qu'à la fin de l'exécution, tout ceux-ci soient verts.

3.2.1 Génération de tests automatiques

Les tests d'intégration du plugin Ford

À court terme, cette solution devrait permettre de tester le plugin pour les projets Ford facilement et de façon efficace. Pour cela, les testeurs Continental vont ajouter des colonnes dans le document *Walkthrough*, afin de spécifier la manière de tester les variables. La plateforme, sera capable d'analyser le document *Walkthrough*, et de générer les tests automatiques. Le testeur n'aura plus qu'à lancer l'exécutable le soir, il reviendra le lendemain, tous les tests auront été exécutés avec un rapport détaillé pour chaque test.

Ces tests s'effectueront sur des variables enregistrées lors de stimulation du contrôleur, afin de vérifier que celui-ci réagit de façon approprié.

Cette plateforme permettra donc de tester facilement la dizaine de projets Ford, et une fois le test d'une variable spécifiée, il n'est plus nécessaire de le réécrire. À chaque nouvelle *release* il suffira de relancer les tests : l'équipe n'aura à faire le travail qu'une fois, ensuite la réutilisation sera possible, les projets seront testés plus rapidement, plus efficacement, et plus souvent.

Les autres projets

À moyen terme, cette plateforme pourrait être utilisée pour les projets d'autres clients tel que Renault, afin d'effectuer là aussi des tests d'intégration. Il était nécessaire de concevoir une plateforme qui puisse évoluer facilement, et avoir un fichier de spécifications en entrée qui soit légèrement différent d'un client à l'autre.

En effet, les autres clients peuvent aussi fournir une partie du logiciel, avec un document de spécification des variables, celui-ci ne serait pas totalement identique, mais l'approche des tests s'en approchera.

Il est également envisageable que la plateforme soit utilisée pour des tests d'intégration en interne, indépendamment des spécifications fournies par le client.

3.2.2 L'utilisation de GreenT comme une bibliothèque

Une autre approche de notre plateforme, serait de s'en servir pour écrire facilement des tests en Java, de façon plus efficace et plus robuste qu'avec la TA3 : notre plateforme doit donc également fonctionner comme une bibliothèque sans utilisation de générateur ou de parser, pour que le testeur puisse effectuer un test rapide.

Celui-ci apprendra à se servir de la plateforme, écrira en règle générale des tests assez courts et moins complexes que ceux que nous générerons, ceux-ci doivent être faciles à écrire.

4 *GreenT* : fonctionnement général

Comme nous l'avons montré dans le chapitre 3, l'entreprise a besoin d'un nouvel outil aidant aux tests d'intégration : *GreenT*.

Nous allons donc voir le développement et la conception de cette plateforme de tests.

Sommaire

4.1 Le fichier Walkthrough 21
4.2 Fonctionnement Global 22
4.3 Les différentes fonctionnalités du Client 24
4.4 Fonctionnement des serveurs 27

Au début de mon stage, le projet ayant un an, les fonctionnalités développées ci-dessous était déjà faites. Je suis cependant intervenu sur la plupart d'entre elles soit pour des corrections de bugs d'une part, soit à des fin d'améliorations d'autres part.

Afin de présenter mon travail que vous trouverez chapitre **??**, il est nécessaire de présenter le fonctionnement général de cette plateforme afin d'en avoir une vue d'ensemble.

4.1 Le fichier Walkthrough

Le fichier Walkthrough est un fichier qui sera fourni par la personne en charge des tests, c'est un fichier au format Excel qui contient les informations de chacune des variables à tester. Il contient ainsi un très grand nombre de colonnes, bien que seule une partie de celles-ci nous intéresse, certaines colonnes ont été remplies par le fournisseur du plugin, d'autres colonnes sont ajoutées dans le seul but de la génération de tests automatiques par *GreenT*. Voici les colonnes les plus intéressantes :

Nom de la variable Le nom de la variable testée : il existe un nom court et un nom long.

Informations aidant à la conversion des données Certains équipements [1] tel que le *debugger* ne fonctionne qu'avec des valeurs Hexadécimales. À la charge de *GreenT* de convertir ces données vers des valeurs physiques exploitables par le testeur. Ces colonnes contiennent les informations nécessaires au calcul de conversion [2].

Nécessité d'un test automatique un `GreenTTest` ne sera généré que si la colonne vaut *Yes*.

Statut du test Nous éditerons cette colonne afin de reporter le statut du test [3].

1. Vous trouverez plus d'informations sur le fonctionnement d'une table de tests section 3.1.1, et dans l'annexe B
2. Informations tel que le domaine de définition physique et le domaine de définition hexadécimal
3. Un test pouvant être Green ou Red mais peut aussi comporter une erreur, tel qu'un problème d'exécution ou de génération, . . .

Precondition (cf section 4.3.2) Contient un scénario d'initialisation du *workbench* : tension de départ, démarrage de l'ECU, ...

Scénario de stimulation (cf section 4.3.2) Contient un ou plusieurs scénarios de stimulations

`ExpectedBehavior` (comportement attendu, cf section 4.3.3) Contient une expression évaluant les variables ayant été enregistrées durant la stimulation : *GreenT* devra vérifier que cette expression est correct à toute instant de la stimulation.

Variable à enregistrer (cf section 4.3.3) Contient les variables devant être enregistrées durant un scénario, en plus des variables présentes dans l'expected behavior. Celles-ci peuvent servir en tant que données contextuelles permettant de mieux cerner le résultat d'un test.

Informations du test (cf section 4.3.5) Plusieurs colonnes tel que la sévérité, le responsable du test, les commentaires, ...

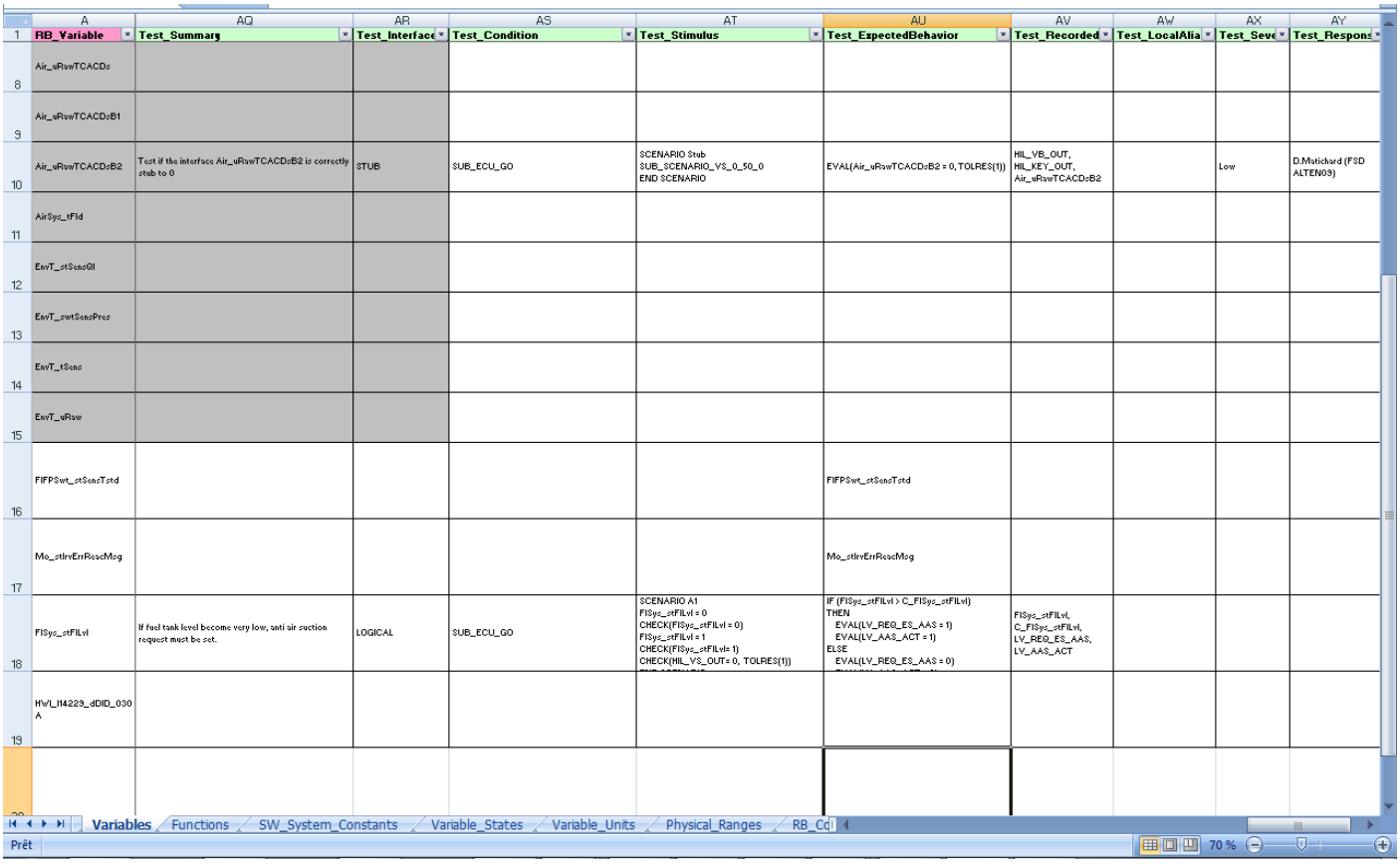

Figure 4.1 – Aperçu d'un fichier Walkthrough – TODO changer la version

4.2 Fonctionnement Global

Le développement de *GreenT* inclu un certain nombre de fonctionnalités attendues par le client et indispensable à son fonctionnement. D'autres fonctionnalités pourront apparaître plus tard en fonction des besoins.

Les principaux modules sont les suivants, avec leurs interactions schématisées figure 4.2 : Dans des objets ovales sont représentés des fichiers, les carrés représentent des modules de la plateforme et

les flèches en pointillés un transfert réseau, les couleurs représentent les différents modules de la plateforme.

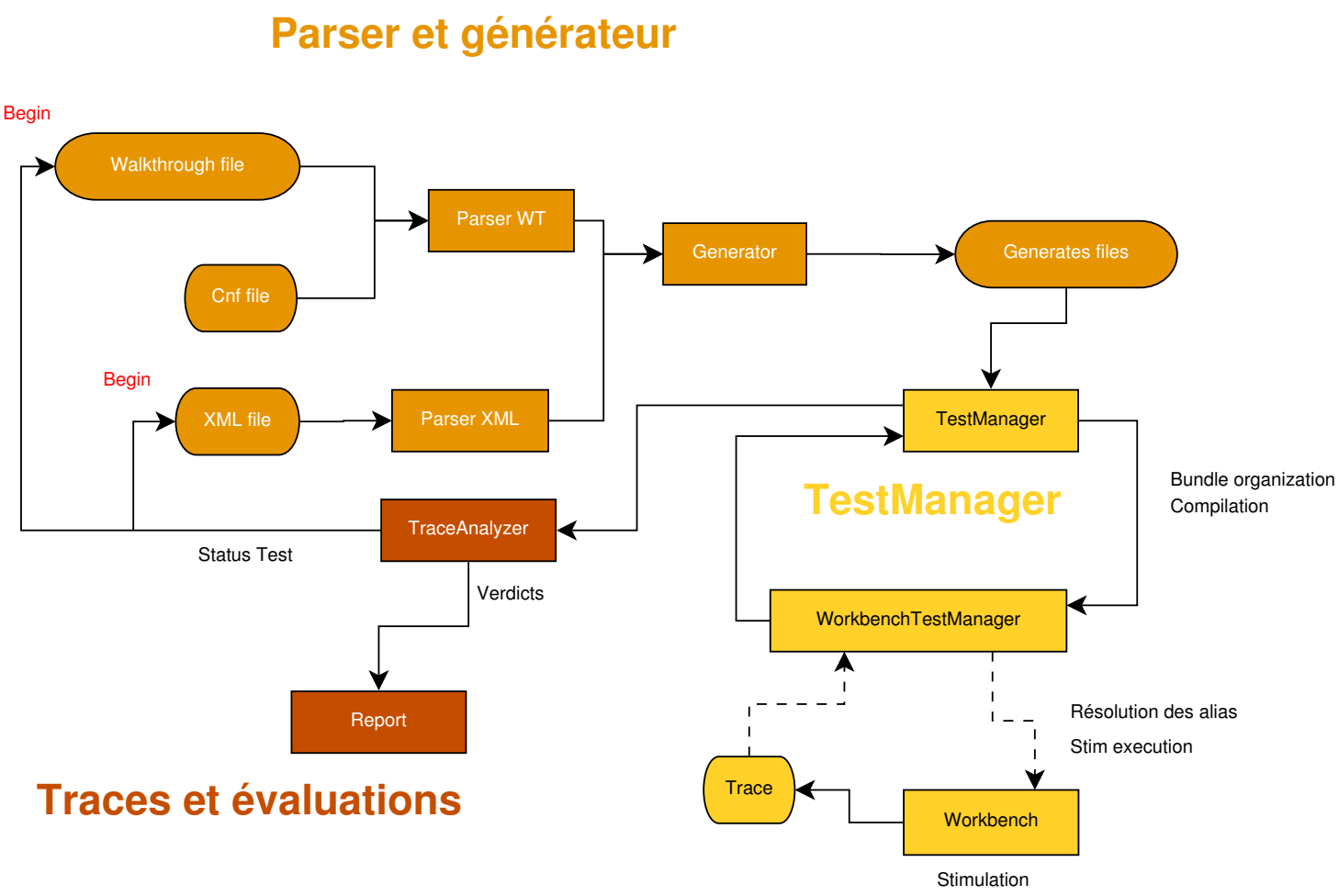

FIGURE 4.2 – Fonctionnement général de la plateforme *GreenT*

Dans la figure 4.2, vous pouvez voir des flèches en pointillés représentant des échanges réseau. En effet l'exécution des stimulations et l'enregistrement des traces se fait par un contrôle distant des bancs de tests [4].

Les principales fonctionnalités demandées par le client sont disponibles sur le client, la majorité peuvent s'effectuer en local :

- Le parsing et la génération (section 4.3.1)
- L'organisation en Bundle (section 4.3.4)
- La production de rapports détaillés (section 4.3.3)

Alors que d'autres fonctionnalités vont nécessiter la présence de serveurs et de connexion réseau permettant d'effectuer ces actions :

- Les stimulations (section 4.3.2)
- Les enregistrement des traces (section 4.3.2)

4. La schématisation du fonctionnement d'un banc est disponible section 3 figure **??**

4.3 Les différentes fonctionnalités du Client

Afin de répondre au mieux aux attentes du client, il a été choisi d'utiliser le Java pour développer notre client, ceci en raison de diverses avantages comme le côté multi-plateforme, son typage fort et sa forte communauté qui permet ainsi une maintenance réduite.

Exemple Afin de comprendre au maximum les tenants et aboutissant de notre plateforme, nous allons utiliser le même exemple tout au long de cette section.

Le testeur souhaite vérifier le bon fonctionnement des ventilateurs de refroidissement du moteur. Nous allons voir de quelle manière il pourrait utiliser *GreenT* afin de vérifier cela.

4.3.1 Parsing et Génération

Le but premier de la plateforme est d'effectuer des tests automatiques, il est ainsi indispensable d'avoir un système d'automatisation.

Pour cela, nous avons un parser : il analyse un certain type de fichier [5] et en retire pour chaque test, le scénario de pré condition, les différents scénarios de stimulations, leurs *Expected Behavior*, les données qui devront être enregistrées ainsi que différentes information sur le test [6].

Une fois toutes ces données acquises, il les transmet à un générateur qui est en charge d'écrire les fichiers Java de chaque test, tous sont organisés dans un dossier temporaire avec un dossier par test. Le `TestManager` peut ensuite traiter ces données.

Exemple Ci-dessous les différentes cellules qui pourrait être renseigné par le testeur pour notre cas d'étude. Nous verrons ce qu'effectuent précisément ces actions dans la suite de la section.

Précondition	Scénario de Stimulation	Expected Behavior

`// Tension batterie`
`HIL_VB = 13;`
`// Clé moteur`
`HIL_KEY = 1;`
`// Vérifications`
`CHECK(HIL_KEY = 1`
`  && HIL_VB = 13);`
	`// Rampe de vitesse véhicule`
	`// de 0 à 50km/h par pas de 1`
	`RAMP(HIL_VS, 0, 50, 0.1, 1);`
	`TODO vérifier syntaxe de RAMP`
		`if temperature > max then`
		`  -- Ventilateur allumé`
		`  EVAL(cfa_on = 1);`
		`else`
		`  -- Ventilateur éteint`
		`  EVAL(cfa_on = 0);`
		`end if`

4.3.2 Stimulation

Afin de tester une variable du plugin, les développeurs vont utiliser des alias présents sur un device : actuellement, un HIL ou un debugger, prochainement nous pourrions en utiliser d'autres que ces deux derniers.

Le spécifieur va rédiger des scénarios de stimulation, ceci afin de mettre le contrôleur dans certaines conditions. Son but sera ensuite de vérifier que ces variables restent cohérentes vis-à-vis du scénario effectué.

5. Nous ne commencerons qu'avec le Walkthrough pour débuter, mais dans le futur nous pourrions avoir des fichiers XML, des bases de données, ...

6. Responsable du test, sévérité, commentaires, nom de la variable, ...

Un scénario particulier doit être spécifié : une pré condition qui a pour but d'initialiser les *devices* et certains alias afin d'avoir un état de stimulation qui soit cohérent et identique à chaque lancement du scénario. Ce scénario sera effectué avant le lancement de chacun des scénarios de stimulation.

Durant l'exécution d'une stimulation, les variables nécessaires sont enregistrées afin de pouvoir produire des rapports et des verdicts ensuite.

Exemple Comme nous l'avons vu section 4.3.1, le testeur nos a donné un scénario de stimulation ainsi qu'un scénario de précondition. Le code généré va ainsi communiquer avec le serveur HIL afin qu'il envoie les bons stimulis à l'ECU.

Ainsi, comme mis en commentaires, le scénario de précondition va mettre la batterie à 13 Volts, et mettre la clé, nous allons ensuite vérifier que cette action a bien été fait. Si tel est le cas, le scénario de stimulations va être effectué. Ce scénario va effectuer une rampe de vitesse, notre véhicule va aller de zéro à cinquante kilomètre-heure avant de retourner à l'arrêt.

4.3.3 Les traces et leurs évaluations

Lorsqu'un scénario de stimulation s'exécute, un certain nombre de variables sont enregistrées : ces variables sont stockées sous la forme d'une trace au format CSV, qui pourra plus tard être représentée sous forme de courbe.

Une fois que la trace est complète, il est nécessaire de l'évaluer : le spécifieur a décrit le comportement attendu dans la colonne *Expected Behavior* détaillant dans quel cas le test est correct, ainsi cette expression va être transformée en arbre logique afin de l'évaluer à tout instant de la trace.

Exemple Durant notre rampe véhicule, le debugger à enregistré différentes variables, en l'occurence nous avons enregistré les trois variables présentes dans notre Expected Behavior : la température du moteur – `temperature`, la temperature maximum sans ventilateur – `max`, ainsi que l'état de notre ventilateur – `cfa_on`.

Ces variables ont été enregistrés durant l'intégralité de la rampe véhicule qui à eu une durée de 40 secondes. À la fin de la stimulation, le serveur retourne ainsi une trace de 40 secondes contenant toutes les variables. *GreenT* peut ensuite évaluer notre *expected behavior* sur l'intégralité de l'enregistrement.

4.3.4 Le module TestManager

Le `TestManager` est le chef d'orchestre de *GreenT*, il a donc un certain nombre de responsabilités.

Il va d'abord organiser les différents tests en un concept que nous avons appelé *Bundle* : afin de limiter le temps d'exécution qui atteindra plusieurs dizaines d'heures, il est intéressant de regrouper les tests possédant les mêmes scénarios de stimulations et les mêmes pré conditions. Seules leur *Expected Behavior* changent, mais celles-ci pourront être évaluées sur la même Trace.

Une fois les tests organisés en *Bundle*, il va les compiler et les donner à un `WorkbenchManager` : toujours pour une raison d'optimisation, il sera intéressant de pouvoir exécuter les enregistrements sur plusieurs bancs simultanément, pour cela le `TestManager` sera capable de savoir quels bancs peuvent être utilisés et pourra distribuer ses *bundles* en fonction.

Chaque `WorkbenchManager` sera en charge d'exécuter le code généré plus tôt et dialoguera en

réseau avec son banc, une fois l'exécution terminée, il obtiendra une trace qui pourra être évaluée.

Afin d'être le plus souple possible, il existe plusieurs modes d'exécution du `TestManager` :

Check only Essaye de parser les différents fichiers, et vérifie que ceux-ci ne comportent aucune erreur de grammaire, d'alias introuvable, d'écriture sur un alias en lecture seule etc...

Parse and generate bundles Parse les fichiers et génère des jars exécutables répartis en bundle

Parse and execute Parse les fichiers, génère les jars pour les bundles et les exécute : c'est le mode « classique ».

Restart test execution Redémarre une exécution qui se serait mal terminée.

4.3.5 Production de rapport détaillé

La plateforme a en charge la production d'un rapport détaillé pour chaque test. Ce rapport contiendra un certain nombre d'informations, et permettra au testeur de comprendre pourquoi le test n'est pas passé. Voici les informations que contiendra ce rapport :

— Nom du test, de la variable à tester
— Nom du responsable du test
— Sévérité du test
— Pourcentage de branches de l'expectedBehavior renvoyant faux(Test « Rouge »), n'ayant pas pu être testé(Test « Gris ») et étant correct(Test Vert)
— Le testeur aura à sa disposition les expressions concernées par un résultat Rouge ou Gris.
— Les colonnes utiles du `Walkthrough`

Actuellement, les rapports se font au format Excel avec l'intégralité de notre enregistrement et pour chaque timestamp, un verdict. Un exemple de rapport est accessible en Annexe D page 43.

Dans un futur proche, ces rapports pourraient être générés dans un format Web avec une possibilité de naviguer entre plusieurs tests, et d'avoir un affichage des courbes de manière graphique.

4.3.6 Mise à jour du Walkthrough

Une fois l'analyse d'un test exécuté, un résultat est mis dans le fichier Excel, en fonction de l'analyse de la trace : un verdict rouge renverra un résultat rouge, si tout est vert le résultat sera vert et enfin, celui-ci pourra être gris si nous n'avons pas été capable d'évaluer de résultats.

> Si il y a eu une erreur à la génération [a] ou à l'exécution [b], la plateforme doit remettre le message d'erreur clair au niveau du test afin que l'utilisateur soit conscient du fait que le test n'a pas pu être exécuté, et pour quelle raison. Libre à lui de corriger le test si nécessaire, ou de le signaler si cela semble être un bogue.
>
> *a.* Mauvaise syntaxe, variables inexistantes, ...
> *b.* Problèmes réseaux, variable non trouvée, communication entre l'ECU et le debugger, ...

4.4 Fonctionnement des serveurs

Comme expliqué précédemment, GreenT va avoir en charge l'exécution de stimulation. Ces stimulations vont communiquer avec une table de tests. Actuellement une table est composée de deux *devices* différents, comportant chacun leur serveur :

- Un HIL, Hardware In the Loop, simulateur d'environnement véhicule
- Un Debugger permettant de voir l'état du programme présent dans l'ECU

TODO : ajouter un schéma Réseau avec GreenT, Serveur DSPace, Serveur T32 et HIL/Trace32
Au début du développement de la plateforme, il a été décidé que les serveurs devront être le plus simple possible pour plusieurs raisons :

- Rabattre le maximum de fonctions métier près du client pour centraliser au maximum le fonctionnement et éviter la maintenance superflue
- Avoir la possibilité de réutiliser les serveurs pour d'autres projets
- Pouvoir ajouter facilement un nouveau device, qui ne nécessiterai que l'ajout d'un nouveau serveur relativement simple

4.4.1 Le serveur Debugger, contrôle de Trace32

Le serveur Debugger propose des services basiques permettant de répondre aux différentes stimulations :

- Flasher le logiciel dans la flash de l'ECU
- Démarrer l'ECU
- Arrêter l'ECU
- Lire une variable ou une calibration
- Modifier une variable
- Enregistrer des variables

Afin d'effectuer ces actions, le serveur s'appuie sur une API fournie par l'outil Trace32 permettant de contrôler le debugger. Ainsi tous nos services vont s'appuyer sur cette API. C'est pour cette raison que ce serveur est développé en Java afin de pouvoir utiliser facilement ces fonctions.

4.4.2 Le serveur HIL, contrôle du DSpace

Le serveur DSpace va devoir lui aussi répondre aux différentes stimulations, ainsi ces services sont relativement similaire :

- Modifier une valeur de la base de données
- Lire une valeur de la base de données
- Enregistrer des valeurs

Ce serveur a été développé en réutilisant une partie de ce qui avait été fait pour la TA3, présenté section 3.1.3 afin de ne pas « réinventer la roue ».

À l'instar du serveur Debugger, nous utilisons une API fournie par l'outil permettant de contrôler le HIL : ControlDesk. C'est ainsi que ce serveur est développé en Python afin de répondre à cette contraintes : l'API du ControlDesk

est en Python.

5 Ma collaboration au projet

Après avoir défini plus en détails les besoins de notre plateforme et son fonctionnement général, nous allons maintenant voir en détail de quelle manière j'ai contribué à ce projet. En parallèle de la maintenance de notre plateforme, j'ai développé deux nouvelles fonctionnalités. Ces deux fonctionnalités ayant un rapport direct avec la notion de « calibration », nous allons tout d'abord définir celles-ci avant de voir en détail mon développement et la maintenance que j'ai effectué.

Sommaire

5.1 Les calibrations 29
5.2 Le « patch calib » 29
5.3 Les « tableaux calibrables » 29
5.4 La maintenance 29

5.1 Les calibrations

Une calibration est une constante stockée en flash, c'est-à-dire en mémoire non volatile. Ainsi le logiciel du contrôle moteur peut accéder à toutes ces calibrations en lecture uniquement.

Ces calibrations permettent de configurer un véhicule avant sa mise en production. Cette configuration peut se faire en fonction de plusieurs critères :

- Le matériel en face du calculateur, comme le nombre d'injecteurs
- La version du logiciel du contrôle moteur
- Leur modification permet de faire une mise au point, permettant d'améliorer la consommation par exemple.

Une fois le logiciel pour un calculateur fourni, et mis en production, ces calibrations ne doivent pas évoluer, les modifier ne sert donc qu'à la mise au point et à la généricité du logiciel.

5.2 Le « patch calib »

5.3 Les « tableaux calibrables »

5.4 La maintenance

Comme expliqué plus haut, j'ai développé deux fonctionnalités durant ce stage. Cependant, en parallèle de ce développement j'ai également corrigés différents bugs, ou améliorer différentes partie de la plateforme.

Ayant conçu cette plateforme lors de mon stage de fin de Licence, je connais l'intégralité de la plateforme ; C'est ainsi que j'ai pu détecter et corriger un certains nombres de problèmes. Ces bugs ou ces améliorations ont été identifiées de trois façons différentes :

— Durant mon développement, il m'est arrivé de trouver du code incohérent ou bouchonné
— Lors d'exécutions de la plateforme sur différentes versions du projet client
— En regardant les différents tickets ouverts et devant être résolus

5.4.1 Corrections

J'ai corrigé quelques problèmes trouvés sur la plateforme, principalement venant du client *GreenT*. Ceci étant dû à notre choix d'architecture : des serveurs les plus légers et un client effectuant le maximum d'actions.

Stockage des erreurs d'exécutions

Le problème En cas d'erreur durant une exécution, si un serveur ne répond plus, si une variable est non trouvée, ... Une exception est levée. À ce moment là, *GreenT* doit attraper l'exception et la stocker en base de données pour pouvoir afficher ensuite le message à tous les tests du bundle, la génération des rapports ne pouvant pas se faire.

La solution Le stockage du message d'erreur n'était pas fait, ainsi que la requête SQL permettant d'obtenir les messages d'erreurs. Ces deux actions ont été corrigés, en lieu et place du rapport de test nous avons maintenant un message d'erreur clair.

Modification des variables Debugger

Le problème Lors d'un stimulus, il est possible de modifier une variable Debugger. Lors de la modification d'une variable, le Trace32 renvoyait toujours une exception, sans appliquer la modification.

La solution Après lecture de la documentation de Trace32, il s'est avéré que le problème venait simplement du serveur qui appliquait une commande syntaxiquement incorrecte. La modification de la commande a corrigée le problème.

Ordre d'exécutions des scénarios

Le problème Un Test peut contenir plusieurs scénarios, ceux-ci nous servent principalement pour le « patch-calib » comme expliqué section 5.2. Or, si nous utilisions plusieurs scénarios, ceux-ci étaient exécutés dans un ordre aléatoire : si l'utilisateur voulait effectuer des actions dans un ordre donnée, ce n'était pas possible.

La solution Le problème avait deux parties : d'une part, l'exécution des scénarios dans un ordre donné, d'autre part, spécifier un ordre à chacun de nos scénarios. En effet, tout d'abord, j'ai nommé les scénarios de sortes qu'ils soient classé par ordre alphabétique. Ensuite, il a fallu spécifier à la plateforme d'exécuter les différents scénarios dans un ordre alphabétique, pour cela il a suffit d'utiliser une collection Java effectuant cette action, la `TreeMap`.

Reset ECU à l'exécution

Le problème Lors de l'exécution de notre plateforme sur la dernière version du projet client, nous avions systématiquement un *reset* ECU. C'est-à-dire que notre ECU s'arrêtait et ne redémarrait pas pour une raison inconnue.

La solution Le problème ne venait pas directement de la plateforme, mais d'une mauvais configuration de notre part. En effet, nous ne spécifions pas les bons fichiers du logiciel, celui-ci étant mal flashé, l'ECU refusait de démarrer.

Absence d'injection

Le problème Lorsque j'essayais de simuler un démarrage du moteur sur la dernière version du logiciel, aucune injection ne se faisait : après le starter, le moteur retournait à 0 tour.

La solution Après s'être renseigné auprès de personnes compétentes, il s'est avéré que cela venait d'un nouveau fichier à flasher dont nous n'avions pas connaissance. Un fichier contenant des calibrations permettant le démarrage du moteur sur table. Ce fichier n'ayant pas été pris en compte durant la conception, j'ai ajouté un nouveau paramètre au fichier de configuration permettant de renseigner des fichiers à flasher additionnels.

5.4.2 Améliorations

Exécution différée

Le besoin Lors de mon développement, j'ai eu souvent des problèmes pour réservés des tables de tests. Celles-ci étant régulièrement prise par les équipes projets.

La solution Afin de ne pas bloquer de tables, et de ne pas bloquer notre travail en raison de l'absence de celles-ci, une solution nous est venue : la possibilité de lancer l'outil durant la nuit. En effet, actuellement une exécution dure environ 45 minutes, après laquelle nous pouvons analyser les rapports et voir les problèmes qui nous sont retournés. Ainsi, j'ai ajouté un nouveau paramètre à l'application permettant de spécifier à laquelle la génération, la compilation, l'exécution et la génération des rapports va se faire. On peut maintenant lancer une exécution le soir et observer les résultats le lendemain matin.

Passage à Java 8

Le besoin La plateforme fonctionnait sous Java 6. Ainsi, nous allions mettre en production une plateforme déjà obsolète à sa sortie. De plus, les deux versions suivantes de Java propose un certain nombre de fonctionnalité aidant au développement, comme des simplifications d'écriture en Java 7 (*Multi-Catch*, Inférence de type, `switch` sur les strings, ...) ou les lambdas-calculs en Java 8. Enfin, dans un future proche nous aurons besoin d'une interface pour *GreenT*, *JavaFx* serait une bonne solutions, mais celle-ci nécessite Java 8.

La solution Avant de passer à Java 8, il a d'abord fallut vérifier qu'aucune incompatibilité avec les bibliothèques que nous utilisons n'allait apparaître. Ensuite, il était nécessaire de télécharger un compilateur ainsi qu'une JVM, configurer les différents environnements et vérifier qu'une exécution se passait de la même manière que précédemment. Après ce succès, le passage à Java 8 a été concrétisé et permet à notre plateforme de rester moderne !

« Clean-Code »

Le besoin *GreenT* ayant deux ans, et ayant connue 4 développeurs différents, il est parfois nécessaire d'améliorer le code existant ou de le rendre plus lisible.

La solution Lorsqu'en développant mes fonctionnalités ou en corrigeant des bugs je tombais sur du code incompréhensible ou du « code mort », je modifiais celui-ci afin de corriger ces défauts. Ceci permet ainsi de garder un code toujours propre et facile à lire.

Affichage des logs

Le besoin La plateforme effectue beaucoup d'actions, allant du parsing jusqu'à la génération des rapports comme montré chapitre 4. Toutes ces actions doivent être tracés, aussi bien en tant réel, en regardant l'exécution que plus tard en observant un fichier.

La solution Les logs fonctionnait déjà, en utilisant `log4j`, cependant celui-ci affichait beaucoup trop d'informations en temps réel, et n'affichait pas les informations les plus utiles. Ainsi, j'ai passé en revue la plateforme pour afficher les bonnes actions (Initialisation des bancs, stimulis effectués, affichage des exceptions, ...). Toutes les erreurs sont redirigé vers la sortie des erreurs (`stderr`), et seules les informations les plus importantes sont sur la sortie console (`stdout`). Tous les autres logs, qui peuvent être utile à la compréhension d'un problème et nous aider ne sont accessible que dans nos fichiers de logs. J'ai par ailleurs ajouté un « buffer tournant » permettant aux fichiers de logs de ne jamais dépasser une certaines taille (5Mio), afin de ne pas consommer trop d'espace.

6 Bilans

Après ce stage de quatre mois, il est temps de dresser un bilan, du point de vue de Continental, ce que mon travail leur a apporté, mais également en quoi ce stage a été bénéfique pour ma future carrière professionnelle.

Sommaire

6.1 Bilan pour Continental 33
6.2 Bilan personnel 33

6.1 Bilan pour Continental

Mon travail dans l'équipe de développement aura été intéressant pour l'entreprise, en partie grâce à ma connaissance de la plateforme suite à mon stage de licence. En effet, avoir contribué au projet l'année précédente sur la conception de celui-ci m'a permis de rapidement commencer le travail et de corriger des bugs répartis dans différents modules. De plus, revenir huit mois plus tard sur ce projet m'a permis d'appréhender le logiciel de manière plus globale et j'ai ainsi pu soulever des problèmes que nous n'avions pas vu lors de la conception.

Grâce à mes connaissances de l'architecture j'ai aidé Benjamin Guerin – le troisième membre de l'équipe arrivant sur le projet – j'ai ainsi pu lui donner des explications et des conseils, pendant que lui apportait un regard neuf à l'existant.

Comme présenté chapitre 5, mon travail aura été directement utile à l'équipe de développement, et au groupe TAS. En effet, j'ai développé deux nouvelles fonctionnalités attendues par le client mais j'ai également corrigé des bugs. Ces corrections de bugs ainsi que les nouvelles fonctionnalités nous permettent maintenant d'exécuter les stimulations ainsi que l'analyse complète sur la dernière version du projet Ford : il est maintenant possible d'avoir les résultats de XX tests en une heure.

Le projet n'est pas terminé, et je n'ai pas pu effectuer toutes les fonctionnalités auxquels nous avions pensés tel que l'utilisation de GreenT avec d'autres fichiers d'entrées que le Walkthrough. Ceci est principalement due à de mauvaises estimations, notamment en raison de la maintenance demandant plus de temps que prévu. Cependant, je vais continuer ce projet dès septembre en contrat de professionnalisation pendant un an et pourrais ainsi finaliser la plateforme.

6.2 Bilan personnel

Annexes

Ci-après vous trouverez un certain nombre d'annexes qui pourront vous permettre de mieux comprendre l'étendue de mon travail et de la plateforme qui est en cours de développement.

Vous y trouverez ainsi un glossaire, des références, des exemples de rapport, de fichiers générés ainsi qu'une explication plus approfondie des équipements utilisés.

A Acronymes et Glossaire

API Application Programming Interface, ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

Antlr Another Tool for Language Recognition, outil permettant de faciliter l'interprétation d'une chaîne de caractère, celui-ci prend en entrée une grammaire, et génère un arbre syntaxique dans plusieurs langages.

Calibration Valeur stockée en flash pouvant contenir une information permettant de simplifier la configuration véhicule. Une calibration pourrait être le nombre d'injecteurs.

ControlDesk Outil permettant de piloter le HIL, l'interface permet ainsi de modifier des valeurs de l'environnement véhicule, ou de pouvoir les lire graphiquement.

Device Les différents équipements dont pourrait avoir besoin l'utilisateur : Hil, Debugger, ...

ECU Electronic Control Unit, calculateur du contrôle moteur

Excel Logiciel tableur appartenant à la suite de Microsoft Office®. Il est possible de modifier une feuille de calcul depuis un logiciel ou un script, notamment en Java.

Flash La mémoire flash est une mémoire de masse non volatile et réinscriptible. Ainsi les données sont conservées même si l'alimentation est coupée.

Flasher Action d'écrire sur la flash, dans notre cas il s'agit d'écrire ou de mettre à jour le logiciel présent sur la mémoire flash de l'ECU.

Grammaire Formalisme permettant de définir une syntaxe clair et non ambigüe.

HIL Hardware in the loop, permet de simuler un environnement véhicule autour du calculateur du contrôleur moteur : celui-ci réagira comme s'il était embarqué dans une voiture.

JAR Java ARchive est un fichier ZIP utilisé pour distribuer un ensemble de classes Java.

Java Langage de programmation orienté Objet soutenu par Oracle. Les exécutables Java fonctionnent sur une machine virtuelle Java et permettent d'avoir un code qui soit portable peut importe l'hôte.

JSON JavaScript Object Notation est un format de données textuelles, générique, dérivé de la notation du langage JavaScript, il permet de représenter de l'information structurée.

JVM Java Virtual Machine

Logiciel de versionnement Logiciel, tel que *Git*, permettant de maintenir facilement toutes les versions d'un logiciel, mais aussi facilitant le travail collaboratif.

Parsing Processus d'analyser de chaîne de caractère, en supposant que la chaîne respecte un certain formalisme.

Python Langage intérprété de programmation multi-plateforme et multi-paradigme. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire et d'un système de gestion d'exceptions.

Apache Thrift Langage de définition d'interface conçu pour la création et la définition de services pour de nombreux langages. Il est ainsi possible de faire communiquer deux problèmes dans deux langages différents : Python et Java dans notre cas.

Trace32 Debugger, permet de debugger un programme embarqué, ceci en permettant de lire la mémoire, mettant des points d'arrêts, ...

UML Unified Modeling Language est un langage de modélisation graphique. Il est utilisé en développement logiciel et en conception orienté Objet afin de représenter facilement un problème et sa solution.

XML Extensible Markup Language est un langage de balisage générique permettant de stocker des données textuelles sous forme d'information structurée.

B Les différents équipements

B.1 Le HIL DSpace

Continental possède différentes tables de tests avec des simulateurs d'environnement véhicule DSPace, comme montré figure B.1.

Figure B.1 – HIL DSpace

Ce simulateur peut être contrôlé via un ordinateur et le logiciel *ControlDesk*. Ce logiciel permet de pouvoir modifier tous les paramètres souhaités de la voiture, comme la tension, la vitesse de rotation du moteur, la mise en place du starter, ...

Une capture d'écran de cette interface est disponible figure B.2.

Figure B.2 – HIL DSpace

B.2 Le Debugger

L'entreprise possède des Debuggers permettant de flasher le programme sur le calculateur, de lire l'état des variables, de modifier des variables, des calibrations, ...

Ce contrôle peut se faire via le logiciel *TLPcasso* dont une capture d'écran est présenté figure B.3. TLP permet d'appeler toute sortes de scripts afin d'effectuer simplement certaines actions.

Figure B.3 – HIL DSpace

C Références

Voici les références que j'ai utilié durant ce stage : particulièrement de la documentation, quelques livres dans lesquels j'ai lu les chapitres qui m'intéressaient pour résoudre un certain problème, certains cours enseignés durant mon cursus m'ont été utiles, et enfin des sites web et forums me permettant de résoudre des problèmes spécifiques.

C.1 Documentations en ligne

Documentation Thrift https://thrift.apache.org/docs/

Documentation Antlr http://www.antlr.org/api/

Documentation Freemarker http://freemarker.org/docs/

Documentation Git http://git-scm.com/documentation

Documentation Java http://docs.oracle.com/javase/6/docs/api/

Documentation Python https://docs.python.org/2.6/

C.2 Livres

UML2 par la pratique – Étude de cas et exercices corrigés sixième édition, 2008. Rédigé par Pascal Roques

Design Patterns : Elements of Reusable Object-Oriented Software Second edition, 1999. Rédigé par le Gang of four : Erich Gamma, Richar Helm, Ralph Johnson et John Vlissides.

The definitive Antlr4 reference 2013, Rédigé par Terence Parr

The LaTeX Companion 2nd édition, 2004. Rédigé par Franck Mittelbach, Michel Goossens, Johannes Braams, David carlisle et Chris Rowley

C.3 Cours magistraux

Design pattern 2015, M1 Informatique Développement Logiciel, Jean-Paul Arcangeli

Modélisation Conception Programmation Orienté Objet 2014, M1 Informatique, Ilena Ober

Langages et automates 2013, L3 Informatique, Christine Maurel et Jean-Paul Arcangeli

Construction et réutilisation de composants logiciels 2014, L3 Informatique parcours ISI, Christelle Chaudet

Conception UML 2012, DUT Informatique, Thierry Millan
Qualité logicielle 2012, DUT Informatique, Thierry Millan

C.4 Sites Web et forums

StackOverflow http://stackoverflow.com
Developpez http://www.developpez.com/
Wikipédia http://en.wikipedia.org/wiki/

D Exemple de rapport généré par GreenT

E Exemples de fichiers générés

E.1 Exemple de StimScenario

```
package com.continental.gt.generation.test.stim;

import java.util.List;

import com.continental.gt.test.stim.StimScenario;

import com.continental.gt.devices.Device;
import com.continental.gt.exception.CheckFailedGreenTTexception;

import com.continental.gt.devices.Hil;
import com.continental.gt.test.alias.AliasHilParam;
import com.continental.gt.test.alias.AliasReadable;

import org.apache.thrift.TException;

/**
 * Test of stubbed class generated
 * Generated by GreenT
 */
public class StimScenario_Stub_1 extends StimScenario {

  public StimScenario_Stub_1(){
    super("Anonymous StimScenario_Stub_1");
  }

  public StimScenario_Stub_1(String name) {
    super(name);
  }

  @Override
  public void addAllRequiredAlias() {
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
    addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
    addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
    addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_KEY"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VB"));
  }
  /*
   * @see com.continental.gt.test.stim.StimScenario#exec()
   * Generated by GreenT.
   */
  @Override
```

```
  public void exec(List<Device> devices) throws CheckFailedGreenTTexception {
    showMsg(".exec() : executing stimulation code of StimScenario_Stub_1 ↩
        class...");
    try {
      double n;
      Thread.sleep(500); // TODO remove me
      Hil hil = (Hil)getDeviceByClass(devices, Hil.class);
      Dbg dbg = (Dbg)getDeviceByClass(devices, Dbg.class);

      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);

      n = getAliasReadable(Hil.class,"HIL_VS_OUT").getValue(hil);
      if(!(n >= -0.2 && n <= 0.2)) {
        throw new ↩
            CheckFailedGreenTTexception("CHECK(HIL_VS_OUT,0.0,TOLRES(1.0))");
      };
      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);

      n = getAliasReadable(Hil.class,"HIL_VS_OUT").getValue(hil);
      if(!(n >= 49.5 && n <= 50.5)) {
        throw new ↩
            CheckFailedGreenTTexception("CHECK(HIL_VS_OUT,50.0,TOLPER(1.0))");
      };
      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);

      n = getAliasReadable(Hil.class,"HIL_VS_OUT").getValue(hil);
      if(!(n >= -0.2 && n <= 0.2)) {
        throw new ↩
            CheckFailedGreenTTexception("CHECK(HIL_VS_OUT,0.0,TOLRES(1.0))");
      };
      dbg.stop();
      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_KEY"))).setPhy(0, hil);
      ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VB"))).setPhy(0, hil);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (TException e) {
      e.printStackTrace();
    }

    showMsg("... complete ok!");
  }

}
```

Extrait de code E.1 – Exemple de StimScenario

E.2 Exemple de GreenTTest

```java
package com.continental.gt.generation.test;

import com.continental.gt.test.GreenTTest;
import com.continental.gt.test.report.Severity;
import com.continental.gt.test.alias.AliasReadable;

import org.apache.thrift.TException;

/**
 * Test of stubbed class generated
 * Generated by GreenT
 */
public class GreenTTest_AIRT_Air_uRawTCACDsB2 extends GreenTTest {

  public GreenTTest_AIRT_Air_uRawTCACDsB2() {
    super("Anonymous GreenTTest_AIRT_Air_uRawTCACDsB2");
  }

  public GreenTTest_AIRT_Air_uRawTCACDsB2(String name) {
    super(name);
  }

  @Override
  public void addAllRequiredContextualData() {
    addRecordedVariable(new AliasReadable("Air_uRawTCACDsB2"));
    addRecordedVariable(new AliasReadable("HIL_VB_OUT"));
    addRecordedVariable(new AliasReadable("HIL_KEY_OUT"));
  }

  @Override
  public void createReport() {
    report.setVariableLongName("Sensed value of down stream charged air ↩
       temperature (bank-2)");
    report.setVariableName("Air_uRawTCACDsB2");
    report.setResponsible("D.Matichard (FSD ALTEN09)");
    report.setSeverity(Severity.LOW);
    report.setTestSummary("Test if the interface Air_uRawTCACDsB2 is correctly ↩
       stub to 0");

    report.addComment("");
    report.addComment("");
    report.addComment("");
  }

  @Override
  public void verdict() {
    // TODO Auto-generated method stub

  }
}
```

Extrait de code E.2 – Exemple de GreenTTest

F Liste des codes sources

E.1 Exemple de StimScenario 45
E.2 Exemple de GreenTTest 47

Table des figures

1.1 Chiffre d'affaire et nombre d'employés (Année 2014) 9
1.2 Répartition du groupe Continental dans le monde 10
1.3 Logo de Continental 10
1.4 Structure de Continental 11

3.1 Fonctionnement d'une table de tests : HIL DSpace, Debugger et ECU 18
3.2 Interfaces du plugin avec le logiciel de Continental 18

4.1 Aperçu d'un fichier Walkthrough – TODO changer la version 22
4.2 Fonctionnement général de la plateforme *GreenT* 23

B.1 HIL DSpace 39
B.2 HIL DSpace 39
B.3 HIL DSpace 40